UPPSALA
UNIVERSITET

U.U.D.M. Project Report 2022:62

Neural ODEs

Andreas Axelsson

Examensarbete i matematik, 15 hp
Handledare: Kaj Nyström
Examinator: Veronica Crispin Quinonez
September 2022

Department of Mathematics
Uppsala University

UPPSALA
UNIVERSITET

Neural ODEs

Examensarbete C i matematik, 15hp

Author: Andreas Axelsson

Supervisor: Kaj Nyström

Department of Mathematics

Uppsala University, August 22, 2022

Abstract

In this paper the concept of Neural networks and Neural ordinary differential equations is presented. This includes both mathematical and machine learning aspects of the subject. With usage of Python, neural networks are implemented with associated algorithms, which are needed for practical experiments. These experiments show great performance in learning the trajectories of two simple dynamical systems, namely a stable spiral ODE and the Lotka-Volterra system. And finally, with the chaotic Lorenz system as input, interesting results arise depending on the architecture and conditions of the network.

Contents

1 Introduction

Integrated technology in human society is more present now than ever. Technical applications and services are available in every sector. It does not matter if it is for finance, healthcare, security or industrial use. All are run by automated systems and share one fundamental principle. Namely to process large amounts of data. To enable the automation of these systems so that human intervention is not needed, some form of artificial intelligence (**AI**) is required. The scientific field in which this is developed goes under the name of machine learning. This is a symbiosis of computer science and mathematics. Neural networks is a framework frequently used in machine learning. A neural network consists of layers, the input layer for receiving data, hidden layers for processing the received data, and lastly the output layer. Deep learning is a subset of machine learning. The "deep" in deep learning is simply indicating to the number of hidden layers in a neural network. If the number of hidden layers exceeds 3 it is considered a deep learning algorithm [4].

A Residual network (**Resnet**) is a neural network with the property of skip connections. Skip connections save the calculated inputs from previous layers separate from the actual output. This is most useful for the purpose of learning. The theory covering machine learning is described more thoroughly in Section 3 and 4. ResNets can be represented by the following function H with parameters

$$H(h_1, h_2, ..., h_{n-1}, h_n, f, \theta),$$

where h denotes the hidden layers and f is the input that is computed through the layers. This network is parameterized by θ in its corresponding layer. The purpose for neural networks is to learn, replicate and recognise the given input. In this paper we will have Ordinary differential equations (**ODE**) representing our input f.

A deterministic dynamical system is described as:

- Some "state" that changes over time.
- Some "rule" for how that state changes over time.
- The rule for how the state changes over time is determined by its differential equations (**DE**)

DEs can be solved by using numerical methods. These methods compute an approximation of the solution with very good accuracy. Numerical methods have been developed and refined since the end of the 1800s and offer a great number of different algorithms [12]. The detailed description of dynamical systems, ODEs and numerical methods resides in Section 2.

By combining numerical methods with neural networks we get a new type of ODEs with properties containing deep learning algorithms. Hence the purpose is to study this symbiosis of numerical analysis and machine learning. More specifically, the purpose of this paper is to study **Neural ODEs** [2].

2 Dynamical systems

Consider f^t to be a dynamical system. f^t poses a group of mappings $M \mapsto M$ which is parameterized by a discrete or continuous time t. Assume that M has a topological and differentiable structure, and that f^t is continuous and differentiable.

2.1 Differential equations

As stated in the introduction, a DE or most common a system of DEs determines the parameterization in time t of the mappings in f^t.

A DE is essentially a condition that consists of functions. The functions describe the rate of change to its primitives. The solution to equations of this form is to differentiate and find these primitives. This can be achieved analytically by integration or Taylor series expansions. Consider the following DE

$$\frac{dy}{dt} = f(t, y). \tag{1}$$

Here f is a given function of two variables. Any differentiable function y = $\phi(t)$ that satisfies equation 1 for all t in some interval is called a solution. DEs splits in two different forms, ODEs and Partial differential equations (**PDE**). A PDE is a DE dependent on more then one variable t. This will include partial derivatives. With Equation 1 as example we find that this include one variable with respect to differentiation. Hence equation 1 is considered an ODE [7].

In the real world a dynamical system describes some phenomenon and is often specified with initial conditions. These conditions includes some interval in time and initial values. This is referred to as an initial value problem. A famous example is Lorenz system of non-linear differential equations [8].

$$\begin{aligned} \frac{dx}{dt} &= \sigma(-x + y) \\ \frac{dy}{dt} &= \rho x - y - xz \\ \frac{dz}{dt} &= xy - \beta z \\ \{\sigma, \beta, \rho > 0 \mid \sigma, \beta, \rho \in \mathbb{R}\}&. \end{aligned} \tag{2}$$

This is a hydrodynamic system and a very simplified simulation of planet Earths atmosphere. And to solve this system one may turn to numerical methods.

2.2 Numerical methods

Numerical methods serve as a powerful tool for evaluating initial condition problems. On fairly simple problems the exact solutions is achievable. However for most problems these evaluations will suffer a marginal change in its outputs due to error in approximations. This is regulated by the step-size which is the number of iterations to a given interval. This is referred to as maintaining *Stability* or numerical *Convergence*. The rate of convergence is measured by the following limit, where x_t is the sequence that converges to the number A.

$$\lim_{t\to\infty} \frac{|x_{t+1} - A|}{|x_t - A|}. \tag{3}$$

To classify the order of convergence we look at the inequality

$$\begin{gathered} \lim_{t\to\infty} \frac{|x_{t+1} - A|}{|x_t - A|^q} < C \\ q \approx \frac{|\frac{x_{t+1}-x_t}{x_t-x_{t-1}}|}{|\frac{x_t-x_{t-1}}{x_{t-1}-x_{t-2}}|}, \end{gathered} \tag{4}$$

where $C < 0$ is some positive constant and q determines the order of convergence [11].

In terms of efficiency this is essential both in regard to the finite memory of a CPU and optimization of the used algorithm. But since the development of numerical methods and its refinement as of today it offers adaptive methods such that the time-step t is optimized for selected accuracy. Therefore memory usage scale with problem complexity [5]

The numerical methods used in this paper is Runge-kutta 4 (**RK4**) and the Dormand Prince method (**Dopri5**). This is an adaptive method in the Runge-kutta family. The algorithm is of the 5th order with 6 stages [3].

3 Neural networks

The motivation behind neural networks is the idea to mimic thought activity of the human brain, implement the brain activity in terms of functions and apply the framework to a computer. Things including recognition are something the brain can do effortlessly. Distinguishing objects, images of objects, sounds such as words and phrases are all in some sense basic abilities and taken for granted.

Remember the representation for some arbitrary neural network 1. A visualisation of the framework may look something like Figure 1

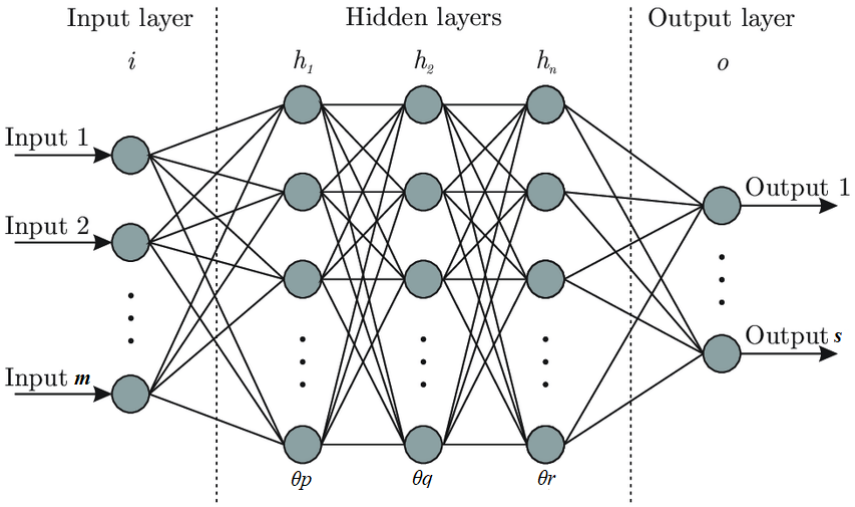

Figure 1: Neural network example

The mathematical implementation of this process from input to output translates to multiplication of matrices. The rows and columns are denoted by connections/weights (the lines) in between each *neuron*(**node**). Every node will represent a unique function inside the main function H. f represents the state of the network before an evaluated layer and bypasses the calculation through that layer. This is called a skip connection. Equation 5 visualizes how the inputs passes through this network.

$$H(h_1, h_2, ..., h_n, f, \theta) = H\left(\underbrace{\overbrace{\begin{bmatrix} i_1 \\ ... \\ ... \\ i_m \end{bmatrix}^T}^{=f_1}}_{=f_0} \begin{bmatrix} m \times p \\ matrix \end{bmatrix}, \underbrace{f_1^T \cdot \begin{bmatrix} p \times q \\ matrix \end{bmatrix}}_{=f_2}, ..., \underbrace{f_{n-1}^T \cdot \begin{bmatrix} r_{n-1} \times r \\ matrix \end{bmatrix}}_{=f_n}\right) = H_{output} \tag{5}$$

This matrix multiplication is squished with a specific function, the Sigmoid or Relu function is a frequent used example. Hence by computation of the inputs through each layer, $H_{output} = L_H - H_{true}$ is the combined evaluation. L_H is called the Loss function, this is the marginal error [6]. Each neuron θ in our network H above is connected to a large number of other neurons. This connection is called *weights*, each neuron inside the hidden layers also come with a *bias*. The bias is a number that indicates if the neuron tends to be active, and adds on to the weighted sum before squishing it with the Sigmoid/Relu function. Therefore this particular network has C connections.

$$C = m \cdot \theta_{p(weight)} + \theta_{p(bias)} \cdot \theta_{q(weight)} + \theta_{q(bias)} \cdot, ..., \cdot \theta_{r(weight)} + \theta_{r(bias)} \cdot o_s.$$

The loss function L denotes the loss of each training example, and serves as an indication if the network is actually learning. A single example is computed by adding up the squares of each difference between the computed outputs and true outputs respectively.

$$loss = (o_{1(pred)} - o_{1(true)})^2 + (o_{2(pred)} - o_{2(true)})^2 +, ..., +(o_{s(pred)} - o_{s(true)})^2.$$

Then by summation of all training examples the average loss will be the output

$$L = mean(loss_1, loss_2, ..., loss_C),$$
$$0 \leq output.$$

Hence L as a function, has C number of inputs and a single number as output, and it is parameterized by the data from training examples. Therefore L is seen as another layer of the network, with zero as optimal output.

3.1 Machine learning

With L as a tool for measurement of learning, one can supervise this when the neural network is running. But how does it actually learn? While L has C inputs it can be written as a vector of length C. By differentiating L, one finds the gradient ∇L which tells in what direction L most rapidly increases. Then naturally $-\nabla L$ is the direction of rapid decrease. Hence by changing the the inputs with regard to the negative gradient repeatedly it descents to a local minimum. This method is called *Gradient descent*. What the network is actually doing when training/learning is to change the activation in each connection so that the average loss will minimize for each block. That is why it is referred to as weights. The algorithm behind this analogy, and what makes it possible is called Backpropagation.

3.2 Backpropagation

With backpropagation the course of computation occurs backwards in order to optimize the weights in previous layers. Consequently the gradient of a hidden layer h_n depends on the gradient of the next layer h_{n+1}.

Theorem 1. *In between two layers i and j in a neural network, the backpropagation algorithm will return the negative partial derivatives with respect to the intermediate weights.*

Proof.

- *Let θ_j be a neuron. θ_1, θ_2,.., θ_m creates outputs o_1, o_2,.., o_m respectively. These outputs are fed as inputs to θ_j.*
- *$w_{1,j}$,.., $w_{m,j}$ are the weights corresponding from o_1,.., o_m onto θ_j.*
- *$f_j = \sum_{i=0}^{j} w_{i,m} o_i = 1$ is the weighted sum of the inputs to neuron θ_j.*
- *$g(x) = \frac{1}{1-e^{-x}}$ is the Sigmoid activation function used by each neuron, such that for θ_j, its output denoted o_j is given by $o_j = g(f_j)$.*
- *x is the vector of input values and y the vector of output values.*
- *h is the function implemented by the network such that $h(x) = \hat{y}$, $h(x)$ is parameterized by the weights.*
- *L is the loss function $L = (y - h(x))^2$.*

Let $w_{i,j}$ be the weight from neuron θ_i to θ_j then by chain rule

$$\frac{\partial L}{\partial w_{i,j}} = \frac{\partial L}{\partial f_j} \frac{\partial f_j}{\partial w_{i,j}}, \tag{6}$$

derive each part separately

$$\frac{\partial L}{\partial f_j} = \frac{\partial}{\partial f_j}(y_j - o_j)^2 = \frac{\partial}{\partial f_j}(y - g(f_j))^2. \tag{7}$$

The j^{th} component of $L = (y_j - o_j)$ and $o_j = g(f_j)$, hence $\frac{\partial L}{\partial f_j}$

$$\begin{aligned} &= -2(y - g(f_j))g'(f_j) \\ &= -2(y - g(f_j))g(f_j)(1 - g(f_j)) \\ &= -2(y - o_j)a_j(1 - o_j). \end{aligned}$$

The second component equals

$$\frac{\partial f_j}{\partial w_{i,j}} = \frac{\partial}{\partial w_{i,j}}(\Sigma_q o_q w_{q,n}) = o_i, \tag{8}$$

the only term that involves $w_{i,j}$ is $o_i w_{i,j}$, therefore

$$\frac{\partial L}{\partial w_{i,j}} = \frac{\partial L}{\partial f_j} \frac{\partial f_j}{\partial w_{i,j}} = -2(y - o_j)o_j(1 - o_j)o_i. \tag{9}$$

Now Δ_j is defined as

$$\Delta_j = \frac{\partial L}{\partial f_j} = -2(y - o_j)o_j(1 - o_j), \tag{10}$$

which implies

$$\frac{\partial L}{\partial w_{i,j}} = -\Delta_j o_i. \tag{11}$$

This is the negative partial derivative of this specific connection. □

This gives the relative amount by how much the network will adjust the weight for activation. The adjustment is regulated with the bias α such that

$$\text{activation}(w_{i,j}) = \alpha \Delta_j o_i.$$

Equation 11 is the negative partial derivative and it serves as one component for the negative gradient. This gives the direction in which the network will progress for better accuracy. With direction this means individual update rules for all weights. From this derivation one can see that the gradient depends on next layer. The proof above is derived with help from [13].

This is why skip connections are important for optimization, since it carries over the data from previous layers. Let say our Resnet consist of 3 hidden layers $H(h_1, h_2, h_3, f, \theta)$. A more conventional way to illustrate the diagram of this network is seen in Figure 2. The rounded pathway on the right hand side illustrates the skip connection.

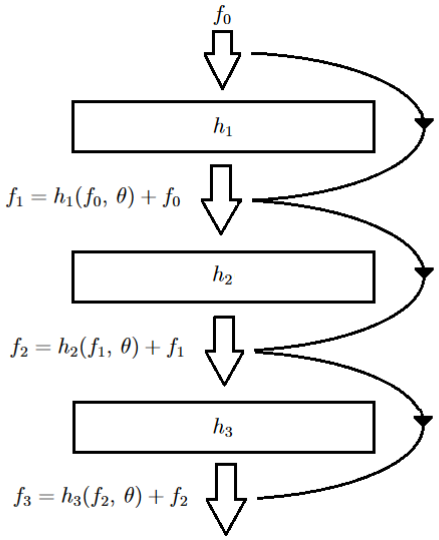

Figure 2: Resnet diagram

4 From Resnet to ODEnet

In order to go from a Resnet to ODEnet one does not leave the Resnet but instead try to replicate it in a way that includes the time t as a parameter. To model a time series like this, the trajectory is determined by the local initial condition f_0 and a global set of dynamics shared across all time series [2]. To achieve this idea, the intermediate layers will be one shared layer h not only parameterized by θ but by t as well. This t now denotes the layers number and therefore at which state in our network the computation occurs. The new function representing our network H is as follows

$$\text{ODEnet } H = H(h, f, t, \theta)$$

With Figure 2 as a reference point, the diagram of our ODEnet shows in Figure 3

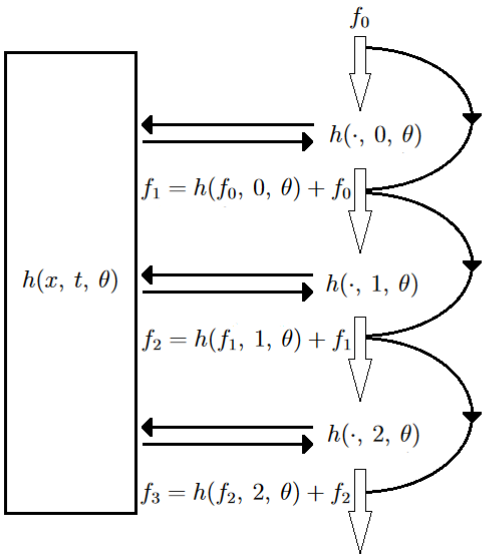

Figure 3: ODEnet diagram

Figure 3 shows an instantiation of H where $t = 2$. This instantiation has three layers h as one shared layer and f_0 as input/initial condition. $h(x, t, \theta)$ now includes the parameter t and $x = f_t$. From the diagram one can see that depending on t the layer function h computes a given state in the network.

Now that H is parameterized by t we can view the output f as a function of t such that $f(t)$

$$\begin{aligned} f_0 &= f(0) \\ f_1 &= f(1) \\ f_2 &= f(2) \\ f_3 &= f(3). \end{aligned}$$

Note that the layers are "hidden" therefore we do not know $f(t)$ inside the network, however H is now a function of t and the only thing needed from f is the initial value $f(0)$. Hence $H(h, f, t, \theta)$ as it is now equals Equation 12 and represents a continuous function. In order to solve this new H ODEsolvers are used so that Equation 13 solves 12, for a given amount of time t_N which determines how many layers are used [2].

$$H(h, f(0), t, \theta) \tag{12}$$

$$\text{ODEsolve}(f(t_0), h, t_0, t_N, \theta_h). \tag{13}$$

In other words, starting from the input layer $f(0)$ as initial condition, the output layer $f(t_N)$ for this new network will be defined as the evaluation of the initial value problem, and this ODE is defined by its dynamical system which is the ODEnet

$$\frac{df(t)}{dt} = h(f(t), t, \theta). \tag{14}$$

Hence the output for the ODEnet is not the actual solution but its derivative, and the loss function L is defined as follows. By treating the result of ODEsolve as input,

$$L(f(t+1)) = L\left(f(t) + \int_t^{t+1} h(f(t), t, \theta)\, dt\right) = L\big(\text{ODEsolve}(f(t_0), h, t, (t+1), \theta)\big). \quad (15)$$

So by moving forward in the ODEnet the trajectory may look something like Figure 4. And everything in between the initial condition $f(t_0)$ and the output $f(t_N)$ is considered a blackbox (hidden layers). By using this method of forward pass the computation is fast but the memory cost is high and additional numerical error emits from the process [2].

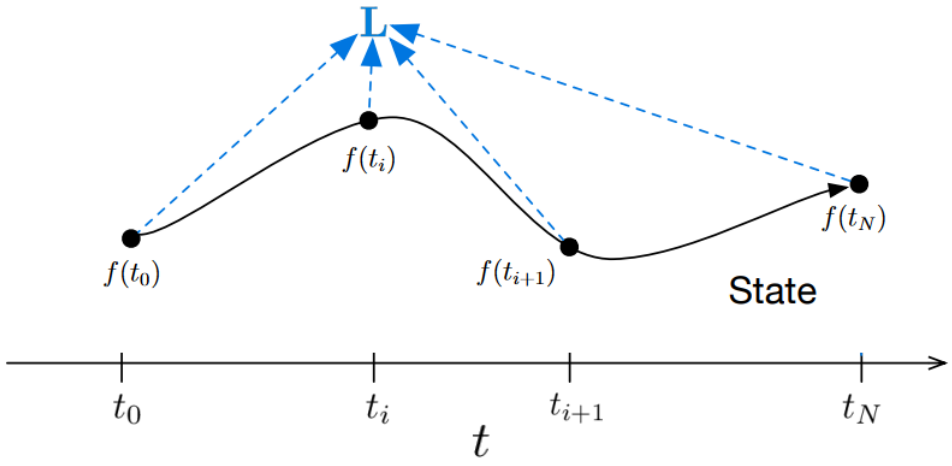

Figure 4: ODEnet state [2]

4.1 Continuous Time Backpropagation

In order to train ODEnets like these with backpropagation the course does not only iterate trough previous layers but backwards in time as well. This means that the initial value problem have to be solved backwards in time in order to follow the trajectory. This includes reverse-mode differentiation through the ODEsolver to find the gradients, and for this the *Adjoint sensitivity method* is used [10].

4.2 Adjoint Method

This method computes the gradients by solving a second ODE backwards in time by treating $f(t_N)$ as the new initial condition and $f(t_0)$ as the requested evaluation from Figure 4. Since the original output from the forward pass equals $L(f(t_N))$ the adjoint initial condition is defined by Equation 16, and the adjoint output by 17

$$a(t_N) = \frac{\partial L}{\partial f(t_N)} \quad (16)$$

$$a(t_0) = \frac{\partial L}{\partial f(t_0)}. \quad (17)$$

Figure 5 offers an illustration of the new trajectory in the adjoint state.

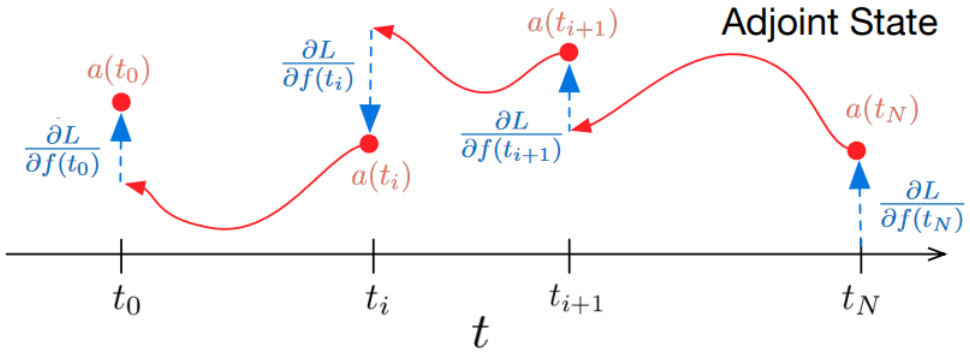

Figure 5: Adjoint state [2]

The adjoint state is defined by

$$a(t) = \frac{\partial L}{\partial f(t)}. \tag{18}$$

The reverse dynamics are given by this second ODE

$$\frac{da(t)}{dt} = -a(t)^T \frac{\partial h(f(t), t, \theta)}{\partial f}. \tag{19}$$

By usage of our ODEsolver we can compute $a(t_0)$ backwards in time starting from $a(t_N)$. The problem that occurs is that we have to know the entire trajectory of $f(t)$ to solve this ODE. Since the method is iterative with respect to t the computation only moves one timestep. Hence $L(f(t))$ together with the adjoint state, this computes from $\frac{\partial L}{\partial f(t+1)}$ to $\frac{\partial L}{\partial f(t)}$ [2]. These iterative steps are shown in Figure 5.

Theorem 2. The adjoint state *a(t)* is the gradient with respect to the hidden layer at a specified time *t*. Then it follows that the adjoint ODE equals

$$\frac{da(t)}{dt} = -a(t) \frac{\partial h(f(t), t, \theta)}{\partial f(t)}. \tag{20}$$

This proof is a modern version of the adjoint method first presented by [10]. All derivations regarding this proof are from the original paper of Neural ODEs [2]. Note that all vector valued expressions in this section with necessary transpose for multiplication has this notation left out for simplicity.

Proof. Note from Theorem 1 in Section 3.2 that in a Resnet the gradient of a hidden layer depends on the gradient in the next layer. By instantiation of the chain rule this translates to

$$\frac{dL}{dh_t} = \frac{dL}{dh_{t+1}} \frac{dh_{t+1}}{dh_t}. \tag{21}$$

As the hidden layers in the adjoint state are continuous, the transformation after an ϵ change in time is written

$$f(t+\epsilon) = \int_t^{t+\epsilon} h(f(t), t, \theta)\, dt + f(t) = T_\epsilon(f(t), t). \tag{22}$$

In the same manner as before, by chain rule we get

$$\frac{dL}{df(t)} = \frac{dL}{df(t+\epsilon)} \frac{df(t+\epsilon)}{df(t)} \iff a(t) = a(t+\epsilon) \frac{\partial T_\epsilon(f(t), t)}{\partial f(t)}. \tag{23}$$

For Equation 20 to satisfy its equality the conviction follows from the definition of derivative

$$\frac{da(t)}{dt} = \lim_{\epsilon \to 0^+} \frac{a(t+\epsilon) - a(t)}{\epsilon}, \tag{24}$$

from Equation 23 we get

$$\frac{da(t)}{dt} = \lim_{\epsilon \to 0^+} \frac{a(t+\epsilon) - a(t+\epsilon)\frac{\partial}{\partial f(t)} T_\epsilon(f(t))}{\epsilon}. \tag{25}$$

Then by Taylor series expansion around $f(t)$

$$\begin{aligned}
&= \lim_{\epsilon \to 0^+} \frac{a(t+\epsilon) - a(t+\epsilon)\frac{\partial}{\partial f(t)}(f(t) + \epsilon h(f(t), t, \theta) + O(\epsilon^2)}{\epsilon} \\
&= \lim_{\epsilon \to 0^+} \frac{a(t+\epsilon) - a(t+\epsilon)\big(I + \epsilon \frac{\partial h(f(t), t, \theta)}{\partial f(t)} + O(\epsilon^2)\big)}{\epsilon} \\
&= \lim_{\epsilon \to 0^+} \frac{-\epsilon a(t+\epsilon)\frac{\partial h(f(t), t, \theta)}{\partial f(t)} + O(\epsilon^2)}{\epsilon} \\
&= \lim_{\epsilon \to 0^+} -a(t+\epsilon)\frac{\partial h(f(t), t, \theta)}{\partial f(t)} + O(\epsilon) \\
&= -a(t)\frac{\partial h(f(t), t, \theta)}{\partial f(t)}.
\end{aligned}$$

□

The similarity in normal backpropagation is present due to the networks architecture. As the ODE for the adjoint state is solved backwards in time. The constraint on the last time point is specified, which is the gradient of the loss with respect to the time point. This is why the gradients of the hidden state can be found at any time, including Equation 16 as initial condition and the initial value given by

$$a(t_0) = a(t_N) + \int_{t_N}^{t_0} \frac{da(t)}{dt} dt = a(t_N) - \int_{t_N}^{t_0} a(t)^T \frac{\partial h(f(t), t, \theta)}{\partial f(t)}. \tag{26}$$

4.3 Augmented State

Now remember that the hidden layers h are parameterized by θ. Optimization of our network is the relative change in θ that minimizes the Loss. In order to compute the gradients with respect to θ we have to evaluate a third integral called the *Augmented state* [2].

We want to obtain the gradients with respect to $\theta - a$ for the loss to reduce. In order to to this θ can be viewed as another component that changes with time and possesses its own dynamics. And since it does not change we define it as a constant state equal to zero

$$\frac{\partial \theta(t)}{\partial t} = 0, \qquad \frac{dt(t)}{dt} = 1. \tag{27}$$

Then by combining both components with our f such that $[f, \theta, t]$, this enables creation of the augmented state. Remember the diagram represented in Figure 3 for sanity check. With the ODEs and adjoint state combined with $[f, \theta, t]$ the augmented state is as follows

$$\begin{aligned} \frac{d}{dt}\begin{bmatrix} f \\ \theta \\ t \end{bmatrix}(t) = h_{aug}([f,\theta,t]) = \begin{bmatrix} h([f,\theta,t]) \\ 0 \\ 1 \end{bmatrix}, \\ a_{aug} = \begin{bmatrix} a \\ a_\theta \\ a_t \end{bmatrix}, \quad a_\theta(t) = \frac{dL}{d\theta(t)}, \quad a_t(t) = \frac{dL}{dt(t)}. \end{aligned} \tag{28}$$

The Jacobian of h_{aug} has the following form with each 0 representing a matrix of zeros with the appropriate dimensions

$$\frac{\partial h_{aug}}{\partial [f,\theta,t]} = \begin{bmatrix} \frac{\partial h}{\partial f} & \frac{\partial h}{\partial \theta} & \frac{\partial h}{\partial t} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}(t). \tag{29}$$

Insert this to Equation 19 to get the following

$$\frac{da_{aug}(t)}{dt} = -[a(t) \;\; a_\theta(t) \;\; a_t(t)]\frac{\partial h_{aug}}{\partial [f,\theta,t]}(t) = -[a\frac{\partial h}{\partial f} \;\; a\frac{\partial h}{\partial \theta} \;\; a\frac{\partial h}{\partial t}](t). \tag{30}$$

Consequently all wanted gradients are acquired. In the first element we find the adjoint ODE 18, the second element obtains the gradients with respect to the hidden θ parameters, by integrating over the full interval we get the total gradient with respect to the parameters

$$\frac{dL}{d\theta} = a_\theta(t_0) = -\int_{t_N}^{t_0} a(t)\frac{\partial h(f(t),t,\theta)}{\partial \theta}dt, \tag{31}$$

and the third element gives the gradients with respect to t_0 and t_N,

$$\frac{dL}{dt_N} = a(t_N)h(f(t_N),t_N,\theta), \qquad \frac{dL}{dt_0} = a_t(t_0) = a_t(t_N) - \int_{t_N}^{t_0} a(t)\frac{\partial h(f(t),t,\theta)}{\partial t}dt, \tag{32}$$

the start and end of the integration interval.

This concludes the mathematical derivation of the Adjoint sensitivity method for reverse mode differentiation of an ODE, and these gradients serve as all possible inputs for our ODEsolver [2].
The algorithm is presented below.

4.4 Full Adjoint Sensitivities Algorithm

Algorithm 1: Complete reverse-mode derivative of an ODE initial value problem

Input: Dynamics params θ, start time t_0, stop time t_1, final state $f(t_1)$, loss $\frac{\partial L}{\partial f(t_1)}$

$\frac{\partial L}{\partial t_1} = \frac{\partial L}{\partial f(t_1)}^T h(f(t_1), t_1, \theta)$ ▷`Compute gradient w.r.t` t_1

$s_0 = [f(t_1), \frac{\partial L}{\partial f(t_1)}, \mathbf{0}_{|\theta|}, -\frac{\partial L}{\partial t_1}$ ▷`Define initial augmented state`

def aug_dynm$([f(t), a(t), a_\theta(t), a_t(t)], t, \theta)$ ▷`Define dynamics on aug state`

return $[h(f(t), t, \theta), -a^T \frac{\partial h}{\partial f}, -a^T \frac{\partial h}{\partial \theta}, -a^T \frac{\partial h}{\partial t}]$ ▷`Compute` ∇`-Jacobian products`

$[f(t_0), \frac{\partial L}{\partial f(t_0)}, \frac{\partial L}{\partial \theta}, \frac{\partial L}{\partial t_0}] = \text{ODEsolve}(s_0, \text{aug\_dynm}, t_1, t_0, \theta)$ ▷`Solve reverse-t ODE`

return $\frac{\partial L}{\partial f(t_0)}, \frac{\partial L}{\partial \theta}, \frac{\partial L}{\partial t_0}, \frac{\partial L}{\partial t_1}$ ▷`Return all gradients`

5 Experiments

In this section experiments will be conducted using the framework of ODEnets and the Adjoint sensitivities algorithm as described above. Implementations are written in Python. Code and documentation for the experiments are in Section 7.

Let us start by examine a system of two linear ODEs

$$\begin{aligned} \frac{dx}{dt} &= \frac{-x}{10} - y \\ \frac{dy}{dt} &= x - \frac{y}{10} \end{aligned} \tag{33}$$

The dynamics are given by the Jacobian matrix

$$J(x, y) = \frac{d}{dt}\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} -0.1 & -1 \\ 1 & -0.1 \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix},$$

with initial conditions

$$f_0 = \begin{bmatrix} 2 \\ 0 \end{bmatrix}.$$

By passing this initial value problem through the ODEnet we find that for each test sample, the loss decreases which increases the learning accuracy. Numerically, we can view this in the table below. This table presents test samples during training, with a frequency of 20 iterations.

t from 0 to 25 for 1000 data points.	
Training sample	Total loss
0020	0.528435
0040	0.255489
0060	0.007868
0080	0.007334
0100	0.005777
0120	0.004273
0140	0.001678
0160	0.001379
0180	0.001127
0200	0.000887
0220	0.001034
0240	0.000611
0260	0.000520
0280	0.000378
0300	0.000345
0320	0.000277
0340	0.000227
0360	0.000235
0380	0.000368
0400	0.000197

Then by plotting the true trajectory with the trained trajectory in the phase plane, the graphical illustration presents itself

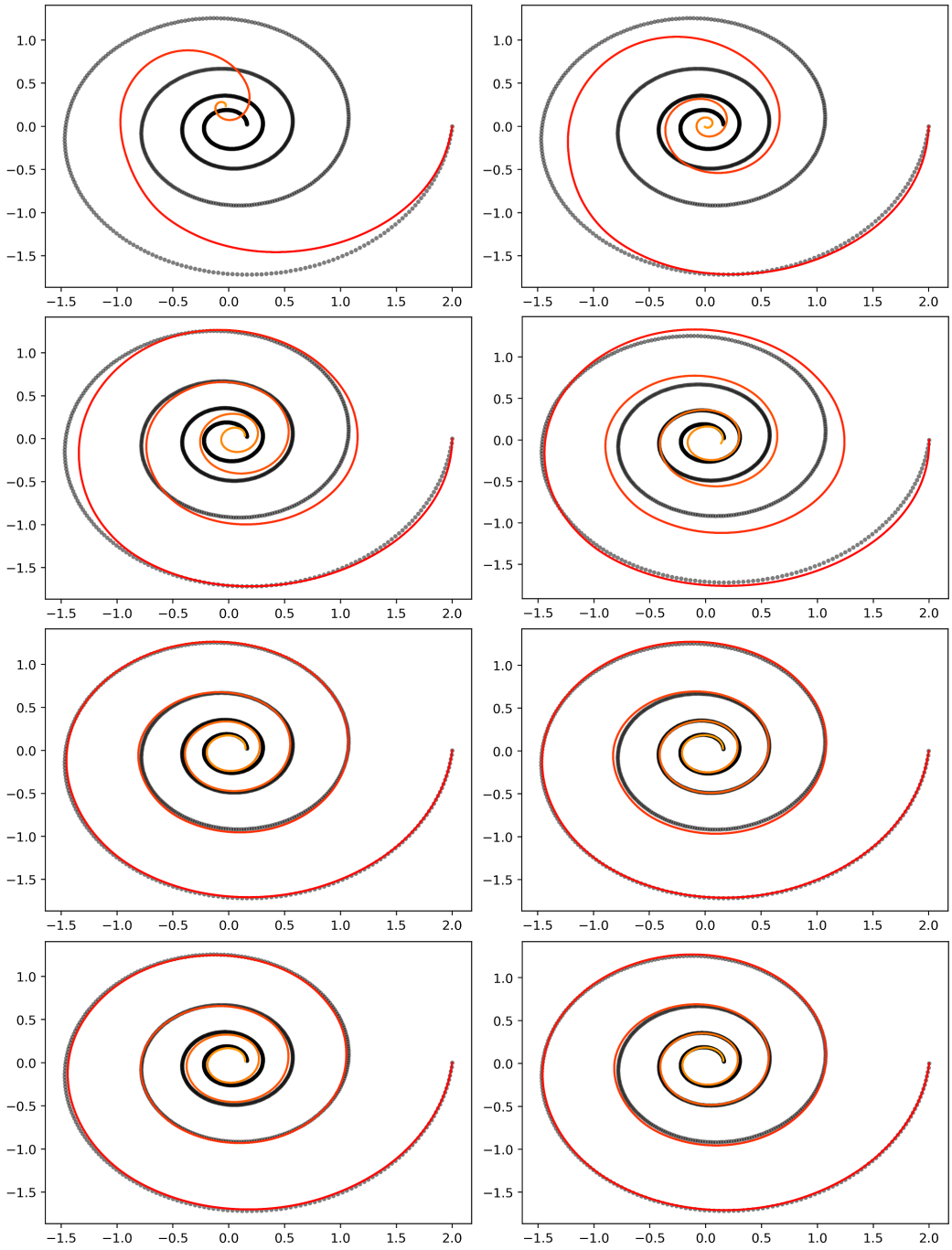

Figure 6: Learning ODE, first and last four samples.

Figure 6 illustrates how the training for f, the orange trajectory increases in accuracy.

Now let us change both initial condition f_0 and time t in Equation 33 for increment in computation complexity. New $f_0 = \begin{bmatrix} 6 \\ 0 \end{bmatrix}$ and t from 0 to 60 for 1200 data points.
With training completed the final output is

Sample	Total loss
Final	0.001089

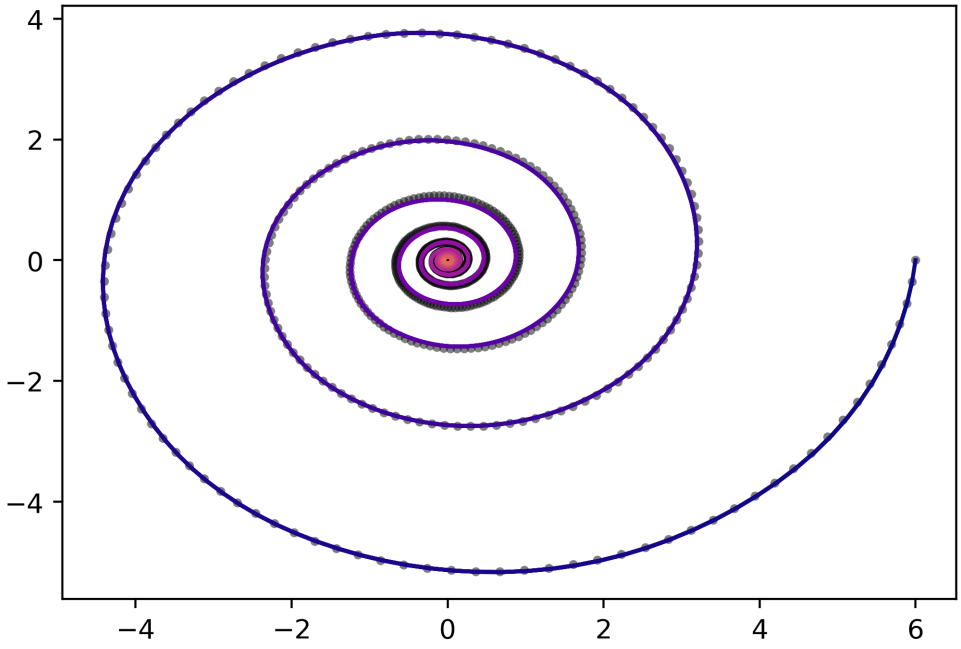

Figure 7: Second spiral

The loss is higher but the ODEnet is still accurate. The properties of Equation 33 make it relatively predictable, it is a linear system which is stable. With time it will converge to a fixed point, in this case origo.

Consider the Lotka-Volterra equations also known as Predator-prey [9]. Unlike Equation 33 this system consists of two nonlinear first order ODEs.

$$\begin{aligned} \frac{dx}{dt} &= \alpha x - \beta xy \\ \frac{dy}{dt} &= \gamma xy - \delta y \\ \{\alpha, \beta, \gamma, \delta &\in \mathbb{R}\} \end{aligned} \tag{34}$$

For $x \neq 0,\ y \neq 0 \implies x = \frac{\gamma}{\delta},\ y = \frac{\alpha}{\beta}$. Therefore the Jacobian matrix is as follows

$$J(x,y) = \begin{bmatrix} \alpha - \beta y & -\beta x \\ \delta y & \delta x - \gamma \end{bmatrix} = \begin{bmatrix} 0 & -\frac{\beta\gamma}{\delta} \\ \frac{\alpha\delta}{\beta} & 0 \end{bmatrix}$$

Let $\alpha = \frac{3}{2},\ \beta = 1,\ \gamma = 3,\ \delta = 1$. This series of plots will illustrate the time space. Learning trajectories are represented as dotted while true ones are continuous and green.

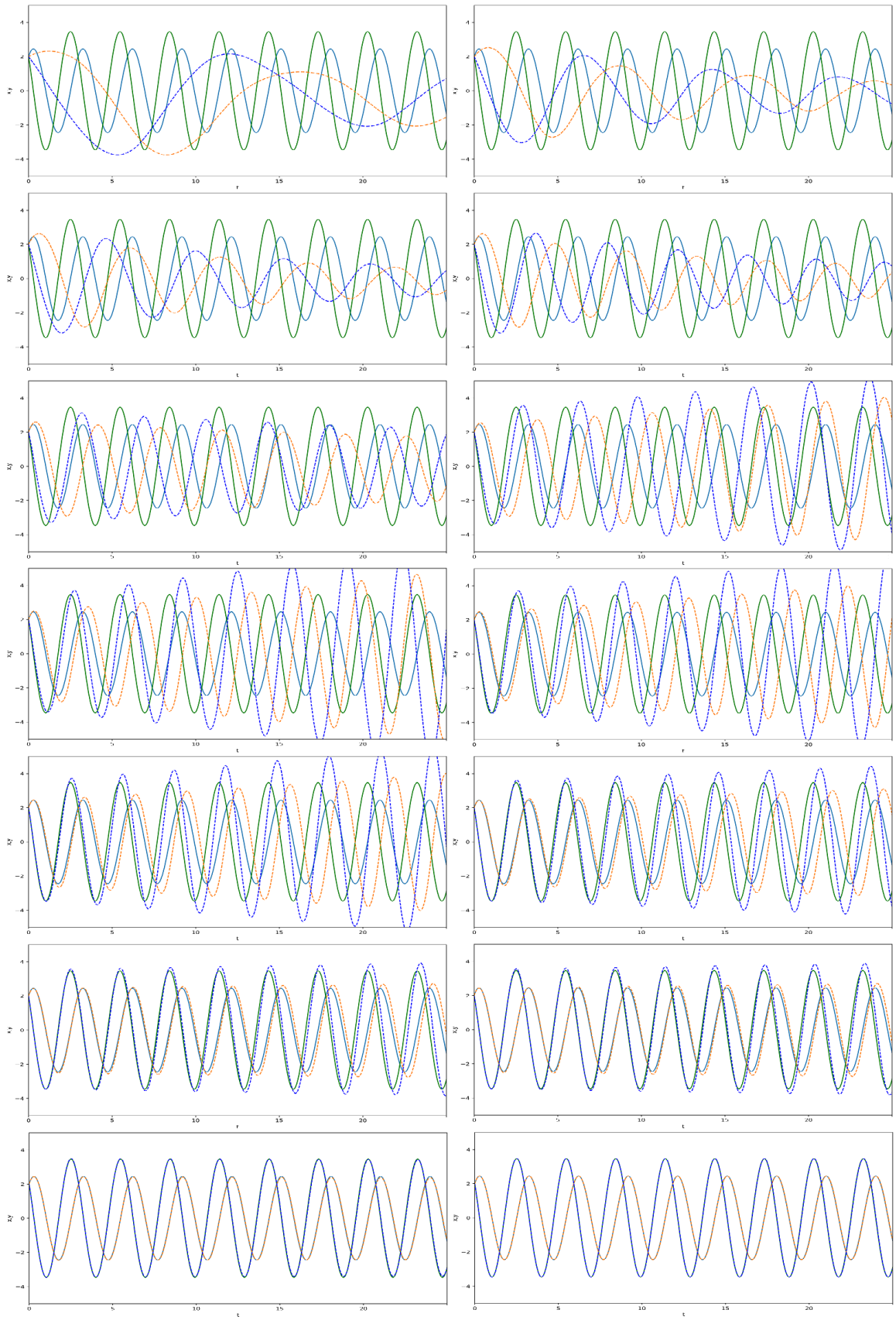

Figure 8: Lotka-Volterra time space, learning samples.

Again the learning evolution for the trajectories is good, with steady decrement in terms of loss. This neural network architecture seems to work great for both linear and non-linear systems. Even if Equation 34 does not converge to a fixed point it is periodic and locally stable.

For the final experiment we will examine the Lorenz Equations 2. The trajectories of this system are neither periodic nor stable. The initial conditions are very sensitive, small changes result in unpredictable outcomes. The trajectories are limited however, and this system is completely deterministic. These properties summarize the structure of a chaotic dynamical system. The Lorenz system is chaotic.

From Equation 2 let $\sigma = 10$, $\beta = \frac{8}{3}$, $\rho = 28$. And set the initial condition f_0 to

$$f_0 = \begin{bmatrix} -8 \\ 7 \\ 27 \end{bmatrix}.$$

This learning evolution is horrible compared to previous experiments. Each test sample appears as if it was random with no steady decrement at all.

t from 0 to 20 for 1000 data points.	
Training sample	Total loss
0020	144.585297
0040	192.734955
0060	197.734024
0080	198.681595
0100	179.537933
0120	162.086349
0140	147.636795
0160	150.344818
0180	193.986435
0200	217.441544
...	...
0500	253.129807
...	...
0800	181.794266
0820	179.595535
0840	179.829285
0860	175.776566
0880	174.230392
0900	180.479126
0920	173.922043
0940	166.760086
0960	160.100464
0980	173.063889
1000	154.976440

The corresponding plots make rapid jumps in the phase space. Sample 500 and 1000 are seen in Figure 9. A closer look at the learned trajectory shows that it is rough.
It resembles straight lines rather then a smooth curve.

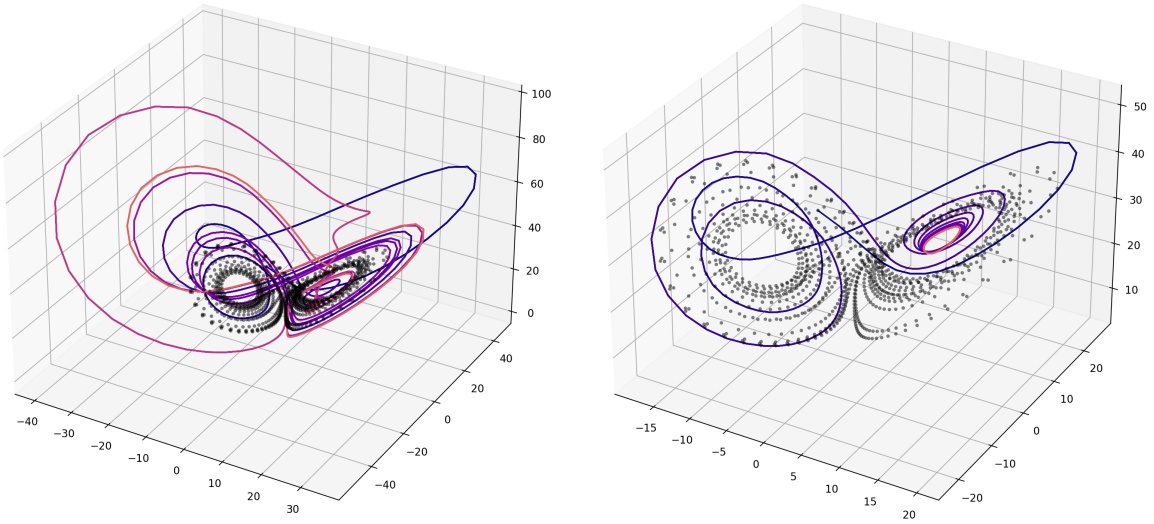

Figure 9: Lorenz attractor sample 500, 1000

Due to the chaotic nature of the Lorenz system, this makes sense. With unpredictable trajectories the interval with respect to time will matter in terms of learning.

After numerous reruns and modifications in the implementation, better results were archived. The revision of foremost significance is the increment in time point density from t_0 to t_N, and increased complexity to the neural network structure. In Section 7 the code for this experiment is the final revision. In this new setting the evaluation time is $t_N = 10$, $t_0 = 0$. The neural network now includes more hidden layers of various proportions. Remember that the number of layers in the ODEnet structure determines in what manner the time series is separated. Now the evaluations will occur with more time points for a shorter time span from $t = 0$ to $t = 10$. The results are presented in the table below.

Table (35)

t from 0 to 10 for 1000 data points.	
Training sample	Total loss
0400	65.436340
0420	78.946884
0440	35.375732
0460	11.629452
0480	11.505157
0500	23.220568
0520	44.665573
0540	36.673378
0560	42.346790
0580	56.542347
0600	20.210516
0620	37.564671
0640	37.265526
0660	36.482655
0680	36.277969
0700	17.576841
0720	21.249786
0740	63.953423
0760	30.317406
0780	40.560822
0800	23.488270
0820	19.991924
0840	4.885432
0860	33.674923
0880	40.415894
0900	165.908981
0920	49.162663
0940	76.682045
0960	42.840584
0980	38.316780
1000	0.716532

The result is far from great, or even good. But it is a major improvement. With original amount of data points kept for the new time interval, the density has doubled. Consequently the curvature of the trajectories shows continuous smoothness. Figure 10 shows plots of the new samples.

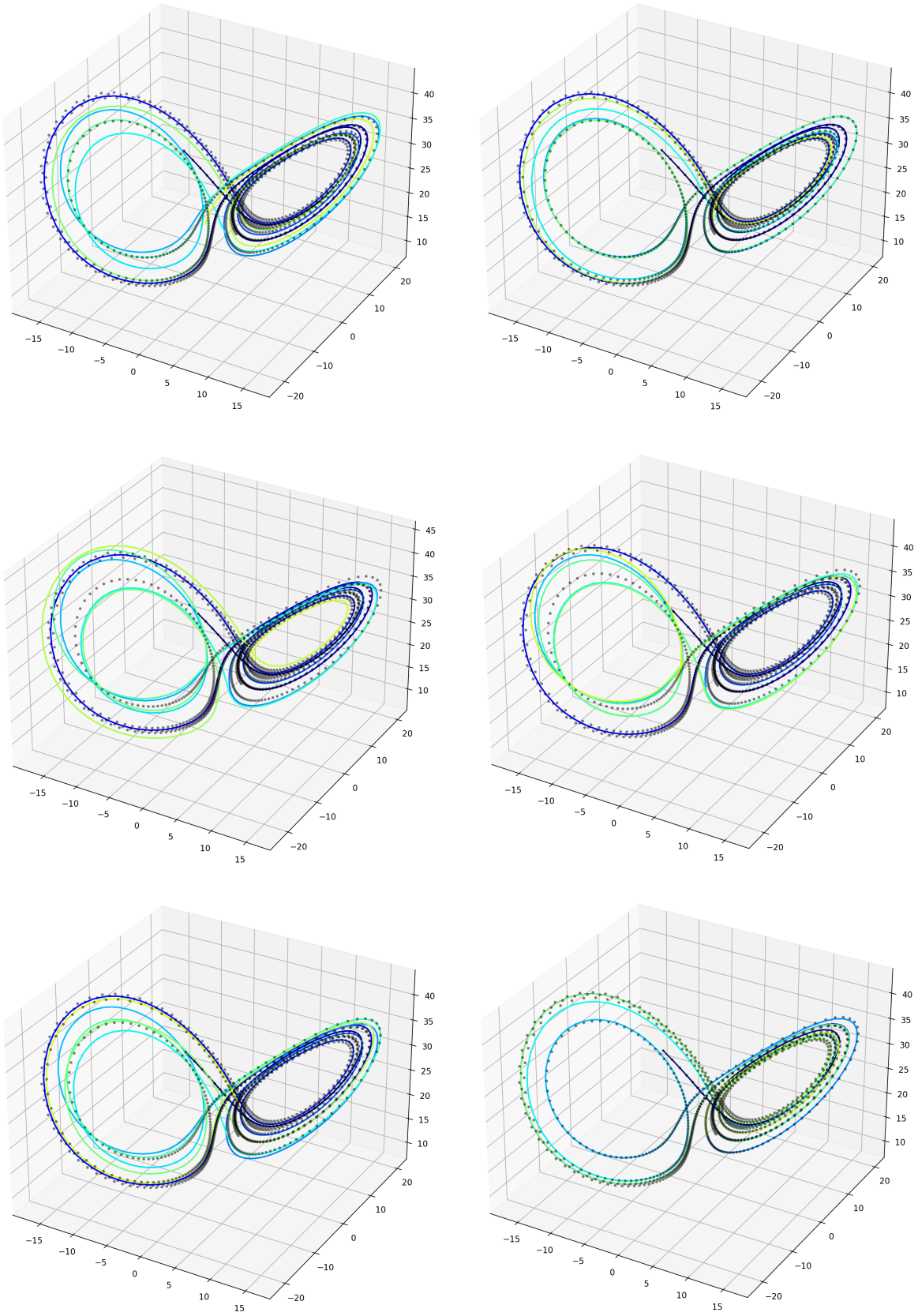

Figure 10: Lorenz attractor sample 0600, 0800, 0940, 0960, 0980, 1000

6 Conclusion

6.1 Results

For simple systems like the linear spiral Equation 33, and Lotka-Volterra Equation 34 this framework implementation displays excellent performance. With stability, predictable behavior and convergence to fixed points, the training data is reliable and efficient. Regardless of initial condition and time interval the network performance is flawless.

With chaotic systems such as Lorenz Equation 2, prediction is non existent for its individual trajectories. What makes *Strange Attractors* interesting is the global stability. Regardless of initial conditions we know with certainty that the butterfly will shape in the phase space [8]. From Table 35 we encounter just that. The evaluated loss for the vast majority of training samples is not anywhere close to zero, while the combined learning trajectory in $\mathbb{R}^3$ resembles the training data really well in comparison. Note that for the first run in the last experiment, Figure 9. It looks like as if the trajectory converges to the center of the right wing. This was taken into account when revisions were made. If the time interval was set to a minimum of $t = 1$ the results would be far more accurate, but then the interest for training dwindle due to short and uninteresting trajectories.

6.2 Further Thoughts

While these experiments exhibit simple examples the adjoint backpropagation is not needed. The increment in numerical error is minimal. Combined with the forward pass seen in Figure 4, normal backpropagation can be used together with ODEsolve for identical results. Still the adjoint method is more reliable due to lower memory cost. For time dependent dynamics of higher complexity such as chemical reactions this is a better choice. With finite memory these calculations blows up really fast while the Adjoint Sensitivities Algorithm enables constant memory usage.

Neural ODEs are a relatively new discovery and a hot topic inside the scientific community of machine learning. With numerous articles released since the original paper of 2018 [2], both limitations and potential uses are presented. From the experiments conducted in this paper the first area that comes to mind is voice recognition. With all new applications for accessibility like smart homes and identification software, security will be of highest importance. If the individual sound waves that people produce could be fitted in ODEs and replicated as its trajectory, then ODEnets would be able to learn these systems with good accuracy, and in turn be able to distinguish if input commands is the voice of the owner or a trespasser.

References

[1] Ricky T. Q. Chen. torchdiffeq, 2021.

[2] Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. Neural ordinary differential equations. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 6572–6583, Red Hook, NY, USA, 2018. Curran Associates Inc.

[3] J. R. Dormand and P. J. Prince. A family of embedded runge-kutta formulae. *Journal of Computational and Applied Mathematics*, 6:19–26, 1980.

[4] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. The MIT Press, 2016.

[5] Ernst Hairer, Syvert Paul Nørsett, and Gerhard Wanner. *Solving ordinary differential equations 1 Nonstiff problems*. Springer, Berlin, 2., rev. ed. edition, 1993.

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[7] E. L. Ince. *Ordinary Differential Equations*. Dover Publications, New York, 1944.

[8] Edward N. Lorenz. Deterministic nonperiodic flow. *J. Atmospheric Sci.*, 20(2):130–141, 1963.

[9] Alfred J Lotka. Contribution to the theory of periodic reactions. *The Journal of Physical Chemistry*, 14(3):271–274, 2002.

[10] L. S. Pontryagin, V. G. Boltyanskii, R. V. Gamkrelidze, and E. F. Mishchenko. *The mathematical theory of optimal processes*. A Pergamon Press Book. The Macmillan Company, New York, 1964. Translated by D. E. Brown.

[11] F. A. Potra. On Q-order and R-order of convergence. *J. Optim. Theory Appl.*, 63(3):415–431, 1989.

[12] C. Runge. Ueber die numerische Auflösung von Differentialgleichungen. *Math. Ann.*, 46(2):167–178, 1895.

[13] Stuart Jonathan Russell and Peter. Norvig. *Artificial intelligence : a modern approach*. Pearson Education Limited, Harlow, 3rd, pearson new international edition. edition, 2014.

7 Appendix

All code written for these experiments uses the Torchdiffeq environment [1]
https://github.com/rtqichen/torchdiffeq

7.1 Code, experiment 1 and 2

This is the main program with all components, for initial value problems of 2 variables.

```
import os
import argparse
import time
import numpy as np

import torch
import torch.nn as nn
import torch.optim as optim
from torch.nn import functional as F
import matplotlib.pyplot as plt
import seaborn as sns
sns.color_palette('bright')


parser = argparse.ArgumentParser()
parser.add_argument('--method', type=str, choices=['dopri5', 'adams'], default='dopri5')
parser.add_argument('--data_size', type=int, default=1000)
parser.add_argument('--batch_time', type=int, default=10)
parser.add_argument('--batch_size', type=int, default=20)
parser.add_argument('--niters', type=int, default=400)
parser.add_argument('--test_freq', type=int, default=20)
parser.add_argument('--viz', action='store_true')
parser.add_argument('--gpu', type=int, default=0)
parser.add_argument('--adjoint', action='store_true')
args = parser.parse_args()


if args.adjoint:
    from torchdiffeq import odeint_adjoint as odeint
else:
    from torchdiffeq import odeint

""" Enables usage of gpu for faster computation """
device = torch.device('cuda:' + str(args.gpu) if torch.cuda.is_available() else 'cpu')
```

```


""" System dynamics """

class Nabla(nn.Module):

    def __init__(self):
        super(Spiral, self).__init__()

    def forward(self, t, y):

        return torch.mm(y, true_dy)


true_y0 = torch.tensor([[2., 0.]]).to(device)  # initial condition
t = torch.linspace(0., 25., args.data_size).to(device)  # time step t_0 to t_N for --data_size
true_dy = torch.tensor([[-0.1, -1.],
                        [1., -0.1]]).to(device)  # Spiral ODE

"""
true_y0 = torch.tensor([[2., 2.]]).to(device)  # initial condition
t = torch.linspace(0., 25., args.data_size).to(device)  # time step t_0 to t_N for --data_size
a, b, c, d = 1.5, 1.0, 3.0, 1.0
true_dy = torch.tensor([[0., -b*c/d], [d*a/b, 0.]]).to(device)
"""

"""Create training data"""
with torch.no_grad():
    true_y = odeint(Nabla(), true_y0, t, method='dopri5')
```

In this code segment you fit the requested ODE. In this case the spiral is active.

```
""" Plot visualization, phase space 2D """

def makedirs(dirname):
    if not os.path.exists(dirname):
        os.makedirs(dirname)


if args.viz:
    makedirs('png')


def visualize(true_y=None, pred_y=None, size=(6, 4)):
    fig = plt.figure(figsize=size)
    if args.viz:

        if pred_y != None:
            z = pred_y.cpu().numpy()
            z = np.reshape(z, [-1, 2])
            for i in range(len(z)):
                plt.plot(z[i:i + 6, 0], z[i:i + 6, 1], color=plt.cm.plasma(i / len(z) / 1.6))

        if true_y != None:
            z = true_y.cpu().numpy()
            z = np.reshape(z, [-1, 2])
            plt.scatter(z[:, 0], z[:, 1], marker='.', color='k', alpha=0.5, linewidths=0, s=45)

        fig.canvas.draw()
        fig.canvas.flush_events()
        plt.savefig('png/{:03d}'.format(itr), dpi=200, bbox_inches='tight', pad_inches=0.1)
```

```
""" Plot visualization, time space """

def makedirs(dirname):
    if not os.path.exists(dirname):
        os.makedirs(dirname)


if args.viz:
    makedirs('png1')


def visualize(true_y, pred_y, size=(10, 6)):
    fig = plt.figure(figsize=size)
    plt.xlabel('t')
    plt.ylabel('x,y')

    plt.plot(t.cpu().numpy(), true_y.cpu().numpy()[:, 0, 0], t.cpu().numpy(),
    true_y.cpu().numpy()[:, 0, 1], 'g-')

    plt.plot(t.cpu().numpy(), pred_y.cpu().numpy()[:, 0, 0], '--', t.cpu().numpy(),
    pred_y.cpu().numpy()[:, 0, 1], 'b--')

    plt.xlim(t.cpu().min(), t.cpu().max())
    plt.ylim(-5, 5)

    fig.tight_layout()
    plt.savefig('png/{:03d}'.format(itr), dpi=400, bbox_inches='tight', pad_inches=0.1)
```

```
""" Neural network implementation for ODE with two variables """


class ODEFunc(nn.Module):

    def __init__(self, y_dim=2, n_hidden=64):
        super(ODEFunc, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(y_dim, n_hidden),
            nn.ReLU(),
            nn.Linear(n_hidden, y_dim)
        )

    def forward(self, t, y):
        return self.net(y)


""" Gradient descent """
def get_batch(traj):
    s = np.arange(len(traj)-1)
    batch_y0 = traj[s]  # (M, D)
    batch_t = t[:2]  # (T)
    batch_y = torch.stack([traj[s + i] for i in range(2)], dim=0)  # (T, M, D)
    return batch_y0.to(device), batch_t.to(device), batch_y.to(device)


""" Computes and stores the average and current value """
class RunningAverageMeter(object):

    def __init__(self, momentum=0.99):
        self.momentum = momentum
        self.reset()

    def reset(self):
        self.val = None
        self.avg = 0

    def update(self, val):
        if self.val is None:
            self.avg = val
        else:
            self.avg = self.avg * self.momentum + val * (1 - self.momentum)
        self.val = val
```

```
""" Training loop """

if __name__ == '__main__':

    ii = 0

    func = ODEFunc().to(device)

    optimizer = optim.Adam(func.parameters(), lr=1e-3)
    end = time.time()

    time_meter = RunningAverageMeter(0.97)

    loss_meter = RunningAverageMeter(0.97)

    for itr in range(1, args.niters + 1):
        optimizer.zero_grad()
        batch_y0, batch_t, batch_y = get_batch(true_y)  # get trajectory batch
        pred_y = odeint(func, batch_y0, batch_t).to(device)  # predict traj with initial_c and time_step
        loss = F.mse_loss(pred_y, batch_y)  # mean squared error
        loss.backward()
        optimizer.step()

        time_meter.update(time.time() - end)
        loss_meter.update(loss.item())

        if itr % args.test_freq == 0:
            with torch.no_grad():
                pred_y = odeint(func, true_y0, t)
                loss = F.mse_loss(pred_y, true_y)  # mean squared error
                visualize(true_y, pred_y)
            print('Iter {:04d} | Total Loss {:.6f}'.format(itr, loss.item()))

            ii += 1

        end = time.time()
```

7.2 Code, experiment 3

This code belong to the previous program. Following segments fit the Lorenz system.

```
""" System dynamics """


class Lorenz(nn.Module):

    def __init__(self):
        super(Lorenz, self).__init__()
        self.sigma = nn.Parameter(torch.tensor([10.0]))
        self.rho = nn.Parameter(torch.tensor([28.0]))
        self.beta = nn.Parameter(torch.tensor([2.66]))

    def forward(self, t, u):
        x, y, z = u[0], u[1], u[2]
        du1 = self.sigma[0] * (y - x)
        du2 = x * (self.rho[0] - z) - y
        du3 = x * y - self.beta[0] * z
        a = torch.stack([du1, du2, du3])
        return a


true_u0 = torch.tensor([-8., 7., 27.]).to(device)  # initial condition
t = torch.linspace(0., 10., args.data_size).to(device)  # time step t_0 to t_N for --data_size
true_u = Lorenz()

""" Create training data """
with torch.no_grad():
    true_fU = odeint(true_u, true_u0, t, method='rk4')
```

```
""" Plot visualization, phase space 3D  """

def makedirs(dirname):
    if not os.path.exists(dirname):
        os.makedirs(dirname)


if args.viz:
    makedirs('png2')


def visualize_3d(true_fU=None, pred_fU=None, trajs=None, size=(10, 10)):
    fig = plt.figure(figsize=size)
    ax = fig.add_subplot(1, 1, 1, projection='3d')
    if args.viz:

        if pred_fU != None:
            z = pred_fU.cpu().numpy()
            z = np.reshape(z, [-1, 3])
            for i in range(len(z)):
                ax.plot(z[i:i + 10, 0], z[i:i + 10, 1], z[i:i + 10, 2], color=plt.cm.jet(i / len(z) / 1.6))

        if true_fU != None:
            z = true_fU.cpu().numpy()
            z = np.reshape(z, [-1, 3])
            ax.scatter(z[:, 0], z[:, 1], z[:, 2], marker='.', color='k', alpha=0.5, linewidths=0, s=45)

        if trajs != None:
            z = trajs.cpu().numpy()
            z = np.reshape(z, [-1, 3])
            for i in range(len(z)):
                ax.plot(z[i:i + 10, 0], z[i:i + 10, 1], z[i:i + 10, 2], color='r', alpha=0.3)

        fig.canvas.draw()
        fig.canvas.flush_events()
        plt.savefig('png1/{:03d}'.format(itr), dpi=200, bbox_inches='tight', pad_inches=0.1)
```

```
""" Neural network implementation for Lorenz system """


class ODEFunc(nn.Module):

    def __init__(self, u_dim=3, n_hidden=256):
        super(ODEFunc, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(u_dim, n_hidden),
            nn.ReLU(),
            nn.Linear(n_hidden, 256),
            nn.ReLU(),
            nn.Linear(256, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 256),
            nn.ReLU(),
            nn.Linear(256, n_hidden),
            nn.ReLU(),
            nn.Linear(n_hidden, u_dim)

        )

    def forward(self, t, u):
        return self.net(u)

""" Gradient descent """
def get_batch(traj):
    s = np.arange(len(traj)-1)
    batch_u0 = traj[s]  # (M, D)
    batch_t = t[:2]  # (T)
    batch_u = torch.stack([traj[s + i] for i in range(2)], dim=0)  # (T, M, D)
    return batch_u0.to(device), batch_t.to(device), batch_u.to(device)
```

```
""" Training loop """


if __name__ == '__main__':

    ii = 0

    func = ODEFunc().to(device)

    optimizer = optim.Adam(func.parameters(), lr=1e-2)
    end = time.time()

    time_meter = RunningAverageMeter(0.97)

    loss_meter = RunningAverageMeter(0.97)

    for itr in range(1, args.niters + 1):
        optimizer.zero_grad()
        batch_u0, batch_t, batch_u = get_batch(true_fU)  # get trajectory batch
        pred_fU = odeint(func, batch_u0, batch_t, method='rk4').to(device)  # predict traj with initial_c and...
      time_batch
        loss = F.mse_loss(pred_fU, batch_u)  # mean squared error
        loss.backward()
        optimizer.step()

        time_meter.update(time.time() - end)
        loss_meter.update(loss.item())

        if itr % args.test_freq == 0:
            if itr > 1800:
                optimizer.param_groups[0]['lr'] = 1e-3
            with torch.no_grad():
                pred_fU = odeint(func, true_fU[0], t, method='rk4')
                loss = F.mse_loss(pred_fU, true_fU)
                visualize_3d(true_fU, pred_fU)
            print('Iter {:04d} | Total Loss {:.6f}'.format(itr, loss.item()))
            ii += 1

        end = time.time()
```